Spring, 1996 Name: ______________________

. My Section Day and Time : ______________________

Com S 342 — Principles of Programming Languages

Test on *EOPL* Chapters 5.6–7, 6.1–3

This test has 8 questions and pages numbered 1 through 7.

Reminders

For this test, you can use one (1) page (8.5 by 11 inches, one (1) side, no less than 8pt font) of notes. No photo-reduction is permitted. These notes are to be handed in at the end of the test. Use of other notes or failure to follow these instructions will be considered cheating.

If you need more space, use the back of a page. Note when you do that on the front.

This test is timed. We will not grade your test if you try to take more than the time allowed. Therefore, before you begin, please take a moment to look over the entire test so that you can budget your time.

For programs, indentation is important to us for "clarity" points; if your code is sloppy or hard to read, you will lose points. We will take off a small amount if you do not give `TYPE` comments for recursive helping procedures. However, you do not have to write such comments for procedures for which the type is stated in the problem. Correct syntax also matters. Check your code over for syntax errors. You will lose points if your code has syntax errors.

Subset of Scheme You May Use

Unless otherwise stated in a problem, when solving problems you may only use standard ADTs used in the interpreter (such as cells, environments, arrays, etc.), standard features of Scheme that we discussed in class, `define-record`, `variant-case`, and helping functions that you define yourself. The standard is defined by the *Revised*^{4} *Report on the Algorithmic Language Scheme*.

Parts of Scheme You May *Not* Use

Unless otherwise stated in a problem, you are prohibited from defining your own macros, and from using internal defines, all the input and output facilities, and the following keywords and procedures. (Don't worry if you don't know what these are.)

```
call-with-current-continuation  do
```

1. (10 points) Using the concrete syntax of the defined language, give the desugared form of the following expression.

   ```
   let x = +(3, 4)
   in *(x, +(x, 5))
   ```

   (Note: your answer must be in the concrete syntax of the defined language; do not write Scheme code.)

2. (10 points) Assuming static scoping, call-by-value and the indirect array model, in the defined language, write a procedure, `arraysubst`, with type

   ```
     (-> (number number (array number)) (array number))
   ```

   that takes two numbers, `new` and `old`, and an array, `arr`, and returns a new array, which is just like `arr`, except that the first occurrence (if any) of `old` in `arr` is replaced by `new`. Note that `arr` is not to be modified. The following are examples.

   ```
   --> definearray a[4];
   --> begin a[0] := 10; a[1] := 11; a[2] := 12; a[3] := 10 end
   --> arraysubst(5, 10, a);
   #(*array* 5 11 12 10)
   --> a;
   #(*array* 10 11 12 10)
   --> arraysubst(7, 12, a);
   #(*array* 10 11 7 10)
   --> arraysubst(7, 14, a);
   #(*array* 10 11 12 10)
   ```

   Your answer should be a completion of the code below. (Hint: you may use `letrec`, `letrecproc`, or a helping procedure. Recall that `arraylength` gives the length of an array and `equal` can be used to compare numbers.)

   ```
   define arraysubst =
   ```

3. (10 points) This is a question about abstract syntax. Recall that, in our notation, the Scheme procedure `parse` as type `(-> (string) parsed-exp)`. For example, the following is an equation between Scheme expressions.

```
(parse "7")
 = (make-lit 7)
```

Complete the following equation, by writing a Scheme expression, using the abstract syntax records for the defined language, that is equivalent to the following.

```
(parse "proc(a, b) +(7, a)")
 =
```

4. (10 points) Briefly answer the following question. What are the (a) advantages and (b) disadvantages of dynamic scope (as compared to static scope)?

5. (10 points) This is a question about dynamic scoping. Consider the following expression in the defined langauge (using call-by-value).

```
let  x = 50; lst = list(3)
in let addlast = proc(lst)
                  if null(cdr(lst))
                  then +(car(lst), x)
                  else addlast(cdr(lst))
   in let x = 4; lst = list(6)
      in addlast(list(7, 9))
```

Using dynamic scoping, (a) Draw a picture of the run-time stack to show how the computation proceeds, and (b) give the result (if any) of the above expression. (If the expression has no result, or encounters an error, write that.)

6. (10 points) This is a question about dynamic assignment. Consider the following expression in the defined langauge (using call-by-value).

```
let green = 1; yellow = 2; red = 3
in let light = green; output = 0
   in let paintlight = proc() output := light
      in begin
            light := red during paintlight();
            list(light, output)
         end
```

Give the result of the above expression.

7. (15 points) Assuming static scoping, consider the following session with the defined-langauge interpreter's read-eval-print loop

```
--> define i = 3;
--> define lst = emptylist;
--> let x = i
    in begin
         i := 77;
         lst := list(i, x)
       end;
```

Fill in the following table with the final values of `i` and `lst` after running the above session, in each of the given parameter mechanisms. (Hint: recall that let is a syntactic sugar.) (If need be, you may use "?" to represent an undefined (i.e., unspecified) value.)

	ending value of	
calling mechanism	`i`	`lst`
call-by-value		
call-by-reference		
call-by-value-result		

8. (80 points) This is a problem about parameter passing mechanisms and array models. Throughout this problem use static scoping. Consider the following expression.

```
letarray a[2]; b[2]
in begin
     a[0] := 5; a[1] := 10; b[0] := 15; b[1] := 20;
     let i = 0; j = 1
     in let f = proc(k,m,u,y,u0,y0,t)
                  begin
                    t := k; k := m; m := t;
                    u0 := +(b[1], j); y := a; a[0] := b[k];
                    %%% draw a picture for this point
                    +(y0,y[0])
                  end
        in let r = f(i, j, a, b, a[i], b[0], 3)
           in list(a[0], a[1], b[0], b[1], i, j, r)
   end
```

For each of the following combinations of parameter passing mechanism and array model: (i) draw a picture of the execution (as discussed in class) for the point noted by the comment, and (ii) give the result of the expression. The combinations you are to do are as follows (there are more on the next page).

(a) Call-by-value with the indirect model.

(b) Call-by-value with the direct model.

Here is another copy of the expression, for your convenience.

```
letarray a[2]; b[2]
in begin
     a[0] := 5; a[1] := 10; b[0] := 15; b[1] := 20;
     let i = 0; j = 1
     in let f = proc(k,m,u,y,u0,y0,t)
                  begin
                    t := k; k := m; m := t;
                    u0 := +(b[1], j); y := a; a[0] := b[k];
                    %%% draw a picture for this point
                    +(y0,y[0])
                  end
        in let r = f(i, j, a, b, a[i], b[0], 3)
           in list(a[0], a[1], b[0], b[1], i, j, r)
   end
```

(c) Call-by-reference with the direct model

(d) Call-by-value-result with the indirect model (copy the results back in left-to-right order)